A Microprogramming Support Tool for Pipelined Architectures

Steven Molnar and Mark C. Surles

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599

Abstract

We describe a software tool to aid the development of microcode for horizontal, pipelined architectures. The tool is a preprocessor for microcode source that allows the programmer full flexibility to optimize code, but removes many of the tedious and error-prone aspects of microprogramming. It automatically allocates floating-point registers, expands complex instructions, and analyzes code for pipeline-related errors.

We have written a working version of the tool for the Weitek XL-8032 floating-point chip set, a horizontal architecture with pipelined sequencer and floating-point datapaths. Although the tool was designed for the XL architecture, the algorithms used are applicable to other parallel/pipelined architectures.

This paper argues for the existence of such tools, summarizes the algorithms needed to analyze control and data flow in the presence of pipelining, and characterizes the tool's performance based on nine microcoded routines written for a real-time 3-D graphics system.

I. Introduction

To achieve higher performances, many of today's VLSI architectures have resorted to instruction and datapath pipelining and wide instruction words. Such architectures offer many MIPs and MFLOPs, but require intricately structured code to utilize them. Many of these processors have had high-level language compilers written for them, often employing sophisticated optimization techniques. Even so, compilers seldom produce code that uses the full instruction width on every cycle. To do so requires that the code be compacted, a difficult process in which code is analyzed for potential parallelism and transformed so that compatible instructions execute together, at the same time ensuring that the original semantics of the algorithm are preserved [9].

Several impressive techniques for approximating optimal code compaction have appeared recently [6,7], though programming by hand with microcode still attains the best performance. Human microprogrammers have access to higher-level information about an algorithm than does a compiler; a human can redesign portions of an algorithm to aid compaction, while a compiler must adhere to the original semantics. As a result, humans can code short routines more compactly than even the best optimizing compilers; complexity becomes an issue only in longer routines where humans have a less clear-cut advantage.

Microcoding by hand, however, is tedious and error-prone. The microprogrammer must contend with low-level details not encountered in high-level languages, making code development significantly slower and more expensive. Many of these details are intrinsic to the goal of optimization, such as the selection of opcodes and the scheduling of instructions. Other details, however, are irrelevant to the task of writing optimal code: allocating variables to registers, checking for latency errors (an insidious problem in pipelined architectures), and writing many copies of similar pieces of code for procedure calls, input/output, etc.

This paper describes an approach used by the authors to speed microcode development on a parallel, pipelined machine by providing the programmer the flexibility to optimize, yet shielding him from irrelevant details. Our effort culminated in the design and implementation of a tool called *MAXL* to aid in microprogramming the Weitek XL-8032 floating point chip set. The tool was used to code portions of a real-time 3-D graphics system. Since the Weitek XL is typical of a large class of VLSI processors, the techniques described here can be used to write similar tools for other machines.

II. The Weitek XL-8032

The Weitek XL-8032 chip set [10] is a horizontally-microcoded, pipelined architecture. It is composed of three separate processors: a sequencer, an integer, and a floating-point processor, all sharing the same 64-bit instruction word. Both the sequencer and floating-point processors are pipelined: the sequencer by one cycle, and the floating-point processor by four cycles (this pipelining is variable under certain conditions). The chip set contains 32 integer registers, 32 floating-point registers, and three temporary floating-point registers. Instructions following a branch (branch shadows) may be executed or not, depending on run-time conditions.

III. Supporting the Microprogrammer

Based on our experience writing microcode for the Weitek XL, we decided that the support tool *MAXL* should have the following features:

- Input should be at the microcode source level. An intermediate or high-level language might make programming easier, but would restrict the programmer's ability to optimize code.
- *MAXL* should automatically allocate floating point registers. Existing code often used 60 or more floating-point variables that were hand-allocated into the 32 floating-point registers. Using *MAXL*, the programmer could define and use symbolic variable names that would

(To be published in MICRO 21: 21st Annual International Workshop on Microprogramming and Microarchitecture (to be held Nov. 29 --Dec. 2, 1988, San Diego, Calif.)

automatically be assigned to registers. *MAXL* would not generate loads and stores to save registers. Instead, it would provide warning messages to the user identifying where allocation was unsuccessful. We assumed the user could do a better job of fixing this than our program.

- *MAXL* need not allocate integer registers; in our experience there has always been a sufficient number of these.
- *MAXL* should accept a superset of Weitek's microcode language. Meta instructions for declaring automatic register variables and specifying procedure calls and debug print statements would be used to supplement Weitek's microcode language.
- *MAXL* should work within single subroutines only. Most code needing to be microcoded can be isolated to a single subroutine. To handle multiple routines would increase the complexity of the tool without greatly increasing its effectiveness.
- *MAXL* should check for as many semantic errors as possible. Of particular importance would be tests for latency violations and unreachable code.
- *MAXL* should be fast enough to be part of the normal program development loop, rather than being considered a "verifier," to be used only occasionally.

IV. Implementing Maxl

To automatically allocate floating-point registers and identify semantic errors, *MAXL* must analyze the semantic structure of the code. Much literature has been written on analyzing a program's control flow and variable usage [1], but most of it assumes that results become available immediately after operations are performed (no pipelining). In designing *MAXL*, we used the algorithms and techniques in the literature as a starting point, but modified them and developed new ones for use in a pipelined domain. *MAXL* is composed of several phases, each abstracting additional information from the raw microcode source until conventional register allocation techniques can be used. Figure 1 illustrates the phases within *MAXL*.

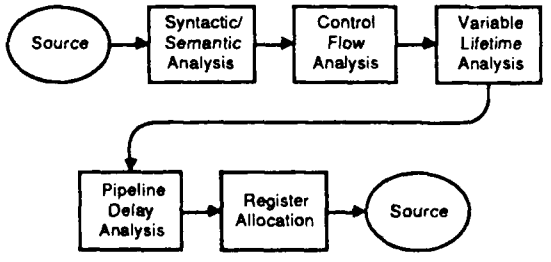

Figure 1. Phases within *MAXL*

The following sections describe each of the phases, giving particular attention to the phases that handle pipelining. Only a summary of the algorithms will be presented here. A more complete description of the algorithms is contained in [4].

Syntactic and Semantic Analysis. This phase parses input and builds a parse tree using the Interface Description Language (IDL) Toolkit [8]. It evaluates semantic attributes, such as jump addresses and variable references. It extracts instruction-dependent information needed in later phases, such as the number and types of arguments. Finally, it determines the conditions under which instructions execute, including the run-time conditional execution of branch shadow instructions.

Control Flow Analysis. This phase propagates control flow information (line adjacencies and branch pointers) to take sequencer pipelining into account. The first step is to connect all sequentially adjacent statements by pointers, except for instructions one and two cycles after an unconditional branch. Next, it adds branch pointers to branch shadow instructions, connecting them to the branch target instruction. Control flow analysis results in a directed graph representing all possible runtime paths within the program.

```
live_dead(instr: instruction;
          Live_in: Set of variables)
begin
   {compute new set of live variables}
   Live_new ← (instr.Live ∪ Live_in ∪
               instr.RHS) - instr.LHS;

   { If instruction has been visited
     before and no change, end recursion }
   if instr.visited AND
      instr.Live = Live_new then
      return
   endif;

   instr.visited ← True;
   instr.Live    ← Live_new;

   {propagate live variables upward}
   foreach path upward in ctrl flow graph
      live_dead(path.instr, Live_new)
   end
end;
```

Figure 2. Recursive routine to compute variable lifetimes

Variable Lifetime Analysis. This phase computes the set of variables, termed *live*, whose values must be retained during each instruction. Optimizing compilers typically do this using data-flow equations [1]. These are solved for transitive closure by repeatedly applying the equations to each instruction in the order of control-flow, beginning at a routine's entry-point. Since we are restricting our analysis to single subroutines, it is more appropriate to begin at the routine's exit point. A recursive *ascent* routine computes variable lifetimes. To implement

this routine we define two additional fields for each instruction: a boolean flag `visited` (initialized to *False*), and the current set of live variables `live` (initialized to {} at each instruction). Figure 2 presents the recursive ascent algorithm `live_dead`.

Pipeline Delay Analysis. This phase propagates variable lifetimes forward through the control flow graph after each assignment to a register variable. This is necessary since interrupts may occur at any time, causing the datapath pipeline to magically empty before the normal latency period has expired. Any live variables sharing the register will be corrupted. Variables being assigned are made live for *latency* instructions following the assignment. This phase also checks for variables that are used before set and handles the special cases where results become available early.

Register Allocation. This phase assigns register variables to physical registers using a graph-coloring algorithm [3]. Only variables that do not interfere may occupy the same register. If register allocation is successful, *MAXL* writes out microcode source containing the new register assignments and expanded meta instructions. If it is unsuccessful, *MAXL* outputs the variables that cannot be assigned and an indication of where the conflict occurred.

V. Results

We began implementing *MAXL* in October, 1987. We released the current version of *MAXL* in April, 1988. Since that time it has been actively used by members the Pixel-planes [5] software team. More microcode routines have been written in the last six months with *MAXL* than were written in the previous year before it existed. The microcoded routines demonstrate a speed-up factor of 3-4 over the same routines written in C. The largest routine written so far uses 78 variables. *MAXL* compressed all 78 into 30 registers. So far no one has written a routine using more floating-point variables than can be allocated into the XL's 32 registers.

VI. Conclusion

MAXL is a microcode development tool that allows the programmer full flexibility to optimize code, while automating many tedious aspects of microcoding. Its success within our group makes us believe that similar tools, carefully differentiating between repetitive analysis best done by software, and creative analysis best done by the programmer, can be useful in other environments and for other architectures. The approach of abstracting information to confine pipeline-related issues to early phases made algorithm development feasible. Since many machines use horizontal instruction formats and pipelining, the ideas contained in this paper could be used to write similar tools for other machines, though every architecture, doubtless, has its own peculiarities. The tool could also be incorporated into the backend of a high-level-language compiler to perform register allocation after parallelizing and code generation.

Acknowledgment

We acknowledge the many people who have contributed to the development of *MAXL*. Trey Greer conceived the idea; Rick Snodgrass provided ideas for algorithms as well as access to the IDL Toolkit; Greg Turk and Vicki Interrante wrote *MAXL's* parsing, and output phases; Clement Cheung currently is supporting the program; and the Pixel-planes team found many bugs in early versions of the program.

This work was supported in part by the Office of Naval Research Contract No. N00014-86-K-0680, NSF Grant No. MIP-8601552, and U.S. DARPA/ISTO Order No. 6090.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison–Wesley, 1986.

[2] A. Aiken and A. Nicolau, "A development environment for horizontal microcode," *IEEE Trans. on Software Engineering*, vol. 14, no. 5, pp. 584–594, May 1988.

[3] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Comput. Lang.*, vol. 6, pp. 47–57, 1981.

[4] C. Cheung, S. Molnar, M. Surles, and R. Snodgrass, "MAXL: A microcode development tool for the Weitek XL-8032," Technical Report (forthcoming), Computer Science Dept., Univ. of North Carolina at Chapel Hill, Chapel Hill, NC.

[5] J. Eyles, J. Austin, H. Fuch , T. Greer, and J. Poulton, "Pixel-planes 4: A summary," *Advances in Graphics Hardware 2: Proceedings of the Eurographics '87 Second Workshop on Graphics Hardware.*

[6] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.

[7] R. A. Mueller, M. R. Duda, P. H. Sweany, and J. S. Walicki, "Horizon: A retargetable compiler for horizontal microarchitectures," *IEEE Trans. on Software Engineering*, vol. 14, no. 5, pp. 575–583, May 1988.

[8] R. T. Snodgrass, *The Interface Description Language: Definition and Use*. Computer Science Press (forthcoming), 1989.

[9] S. R. Vegdahl, "Local code generation and compaction in optimizing microcode compilers," Ph.D. dissertation, Dept. Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, December 1982.

[10] Weitek Corporation, *XL-Series Hardware Designer's Guide*. Weitek Corporation, Sunnyvale, CA 94086, December 1987.